ML 02: GRADIENT DESCENT

Machine Learning by Andrew Ng, *Coursera*

Linear Regression With Multiple Variables

Mutiple Features

변수가 적을때는 *Hypothesis* 가 간단하다. 많으면 어떻게 될까? *Feature* 가 `N+1` 개라면,

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n$$

http://bt22dr.wordpress.com

편의상 `x_0 = 1` 이라 두면, *Hypothesis* 는 *Zero-based index* 인 `n+1` 벡터 `h` 와 `x` 의 곱이다. 따라서 `h(x) = h_t * x` 로 표기할 수 있다. 이걸 **Mutivariate linear regression** 이라 부른다.

Gradient Descent for Multiple Variables

Cost function 은 다음과 같다. 변수의 subscript 는 `j` 번째 *Feature* 를, superscript 는 `i` 번째 데이터임을 말한다.

$$J(\theta_0,\theta_1,\ldots,\theta_n) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

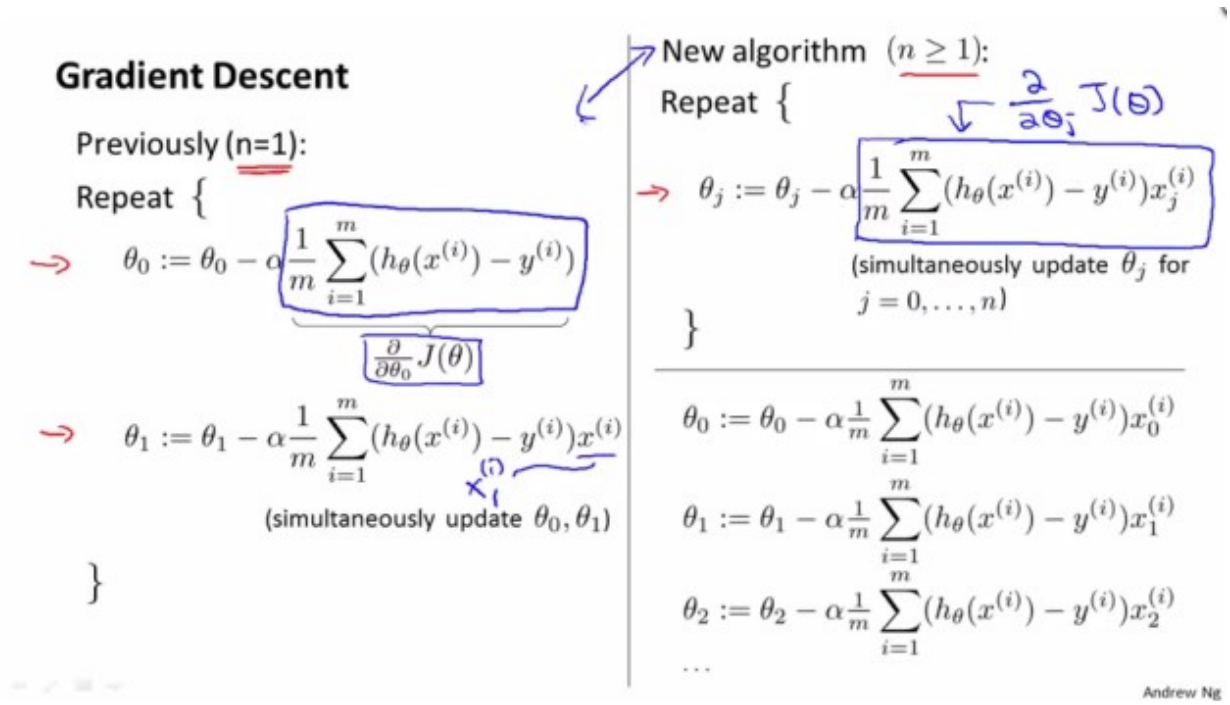
(http://bt22dr.wordpress.com/)

다음은 *Gradient Descent* 알고리즘을 구하는 정의다.

$$\begin{aligned} &repeat\ \{ \\ &\quad \theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta_0,\theta_1,\ldots,\theta_n) \\ &\} \end{aligned}$$

(http://bt22dr.wordpress.com/)

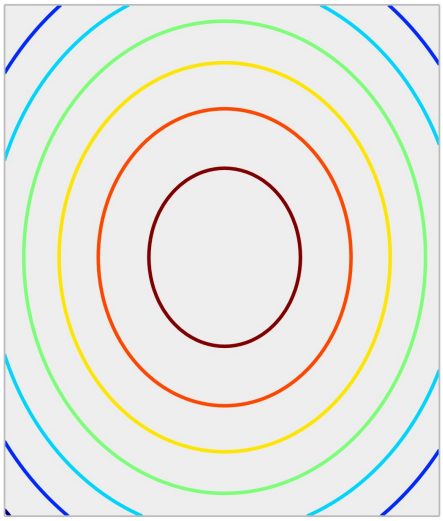
따라서

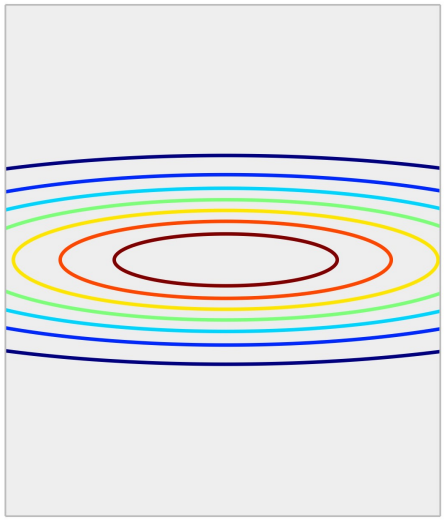


(http://bt22dr.wordpress.com/)

Feature Scaling

Feature 간 데이터 크기가 많이 차이가 나면, *Gradient Descent* 에서 등고선 간 간격이 좁으므로, *Global optima* 를 찾는데 오래걸릴 수 있다. 따라서 *Feature* 값을 `m` 으로 나누거나 -1 과 1 사이로 *scaliing* 할 수 있다. 거꾸로 말하면, *Feature scaling* 을 이용하면 *Gradient descent* 가 결과값을 더 빠르게 찾는다.





또한 **Mean normalization** 을 이용할 수 있는데, 모든 *feature* 에서 평균을 빼서, 평균을 0 으로 만드는 방법이다.

더 일반적인 방법은 *mean normalization* 을 하고, 거기에 `max-min` 또는 *standard deviation* 으로 나누는 방법이다.

Learning Rate

디버깅 팁 중 하나는, 우리가 작성한 *Gradient descent* 알고리즘이 매 *interation* 마다 줄어들어야 한다는 것이다.

(http://spin.atomicobject.com)

그리고, 어느 지점에선가 *converged* 되는지 검사하기 위해 *automatic convergence test* 를 사용할 수 있다. 예를 들어 한 이터레이션에서, 10^-3 보다 적게 줄어드는지 검사한다거나.

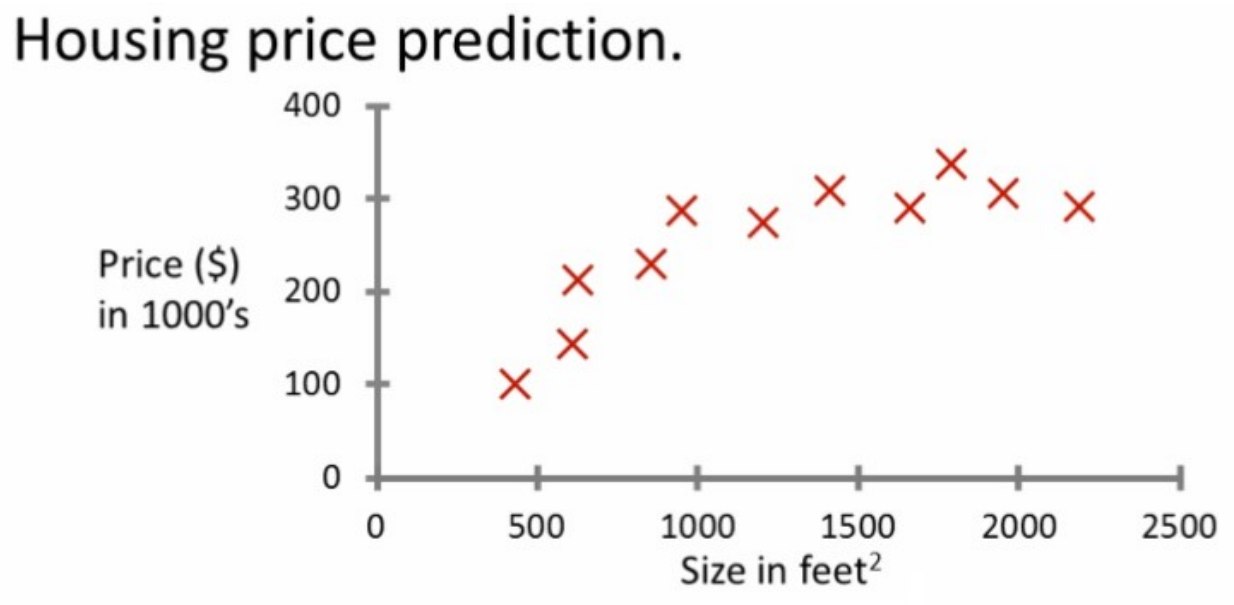
만약에 *gradient descent* 값이 증가하면, 더 작은 *learning rate* 를 사용해라. 그렇다고 너무 작은 값을 사용하면 *gradient descent* 가 느리게 수렴할 수 있다. *learning rate* 가 너무 크면, 심지어 수렴하지 않을 수도 있다.

따라서 *learning rate* 를 `0.001`, `0.003`, `0.01`, `0.03`, `0.1`, `0.3`, `1` 처럼 작은 것부터 선택하되, 천천히 늘려가는 것이 좋다.

Polynomial Regression

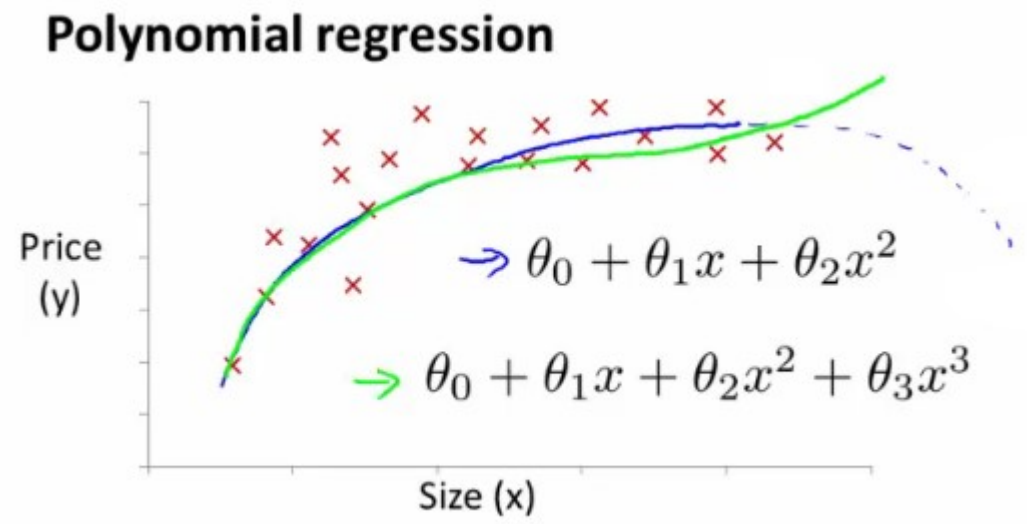
집값을 예측하기 위해 두개의 *feature*, `frontage` 와 `depth` 가 있다고 하자. 두 값을 곱해 `area` 라는 새로운 *feature* 를 만들면, *Hypothesis* 가 간단해진다. 따라서 기존의 *feature* 를 이용 할 수 있는지도 잘 알아보는 게 좋다.

자 이제, 집 값(Housing prices) 을 예측하기 위해 *Size(Area)* 라는 *feature* 를 이용한다 하자. *training set* 이 다음과 같을때,



http://www.holehouse.org/mlclass

hypothesis 를 *quadratic* 로 세우면 어느 지점부터는 예측된 값이 감소하므로 *traning set* 과 일치하지 않는다. 따라서 *cubic* 다항식을 이용해 볼 수 있겠는데, *feature* 가 `size` 하나 뿐이므로, *hypothesis* 는 `size` 를 이용한 삼차식이 되겠다.



http://www.holehouse.org/mlclass

이 경우 `size` 하나로 3개의 *feature* 를 만들었으니, *scaling* 이 문제가 될 수 있다.

이 전에 앞서서 *feature* 가 두개인 *hypothesis* (quadratic) 은 말이 안된다고 했는데, 두개지만 *square* 모델을 사용하면 우리가 가진 *training set* 과 얼추 맞아 떨어지는 모델을 찾을 수 있다. 그림이 없어서 대충 식을 첨부하면,

```
h(x) = y0 + y1(size) + y2 * square(size)
```

여기서 `y` 는 강의에서 말하는 `0(theta)` 라 보면 된다.

Nomal Equation

gradient descent 는 반복하면서 특정 값에 수렴해 가는 알고리즘 이었지만 **normal equation** 은 그냥 `J(0)` 식을 풀어버려 값을 찾아낸다.

예를 들어서 `J(0)` 가 `0(theta)` 에 대해 *quadratic* 이면, `0` 에 대해 미분해서 최저점을 찾아내면 된다. 문제는, `0` 가 여러개 일때, 모든 `0_j` 에 대해 *cost function* 을 풀어야 한다는 것이다. *partial derivative* 를 이용해서 해를 찾으면 된다.

(http://www.longhaiqiang.com/)

행렬을 이용할 수도 있다. 자세한 건 강의 내용을 보자, *design matrix* 라고 부르는 `X` 를 만들어서 아래의 식을 구하면 된다. 사실 `X` 는 그냥 *feature* 들을 있는 그대로 행렬로 만들면 된다. 맨 앞에 `x0` 만 추가해서.

(http://www.longhaiqiang.com/)

참고로, 저 식을 *Octave* 에서는 다음과 같이 계산한다.

```
pinv(X`*X)*X`*y
```

normal equation 을 이용할때는 *feature scaling* 을 하지 않아도 괜찮다. *gradient descent* 와 비교해 보자면,

> **Gradient Descent:**
> (1) *learning rate* 를 골라야 한다. (2) *feature scaling* 을 해야할 필요가 있다.
> (3) *interation* 을 해야하므로 알고리즘이 제대로 돌아가는지 체크해야할 필요가 있다.
> (4) 대신 `n` 이 커도 잘 돌아간다.
>
> **Normal Equation:**
> (1) *learning rate* 를 고를 필요가 없다.
> (2) *feature scaling* 을 해야할 필요가 없다.
> (3) *interation* 을 하지 않는다.
> (4) `n` 이 커질경우 굉장히 느려지고 `(X^TX)^-1)` 을 계산해야 한다.

따라서 `n` 이 너무 크지 않으면, 100~1000 정도까지는, *normal equation* 을 쓰는편이 낫다.

Nomal Equation Noninvertibility

만약에, 우리가 가진 `X` 가 *non-invertible* 하다면 어떻게 될까? *invertible matrix* 란, 아래를 만족시키는 `B` 가 존재하는 행렬이다. `I` 는 *identity matrix* 다.

$$\mathbf{AB} = \mathbf{BA} = \mathbf{I}_n$$

(http://en.wikipedia.org/wiki/Invertible_matrix)

만약 저런 `B` 가 존재하지 않아 *non-invertible* 한 행렬을 **sigular matrix**, **degenerate matrix** 라 부른다.

우리가 계산해야 할 행렬이 *non-invertible* 이라면, 두 가지 경우가 있을 수 있는데,

(1) Redundant features(linearly dependent) e.g `x1 = (3.28) * x2`
(2) too many features e.g `m <= n`

이럴 때는 몇몇 *feature* 를 삭제하고, *regulaization* 을 하면 된다.

Cost Function: Octave

cost function 을 구현 해 보면

```
function J = costFunctionJ(X, y, theta)

m = size(X, 1) % number of training examples
predictions= X * theta; % predictions of hypothesis on all m examples
sqrErros = (predictions-y).^2;

J = 1 / (2*m) * sum(sqrErros);
```

R 이나 이런것들은 행렬연산이 참 쉬운것 같다.

Vectorization

Vectorization 을 이용하면, `for loop` 을 제거할 수 있는데, 예를 들어

이건 행렬 곱셈이 한번에 이루어진다는 것을 이용한 방법이다. 따라서 *gradient descent* 알고리즘에서 `theta` 를 `for-loop` 으로 구하는 것이 아니라, *vectorization* 을 이용하면 한번에 계산할 수 있다.

이게 그림을 구하기가 어려운데, 아래첨자(sub-script) 를 이렇게 기술한다고 하자. `x_0` 그럼, *grandient descent* 알고리즘 식에서 *learning rate* 뒷부분이 *vector* 가 되는데 그 이유는 `theta` 와 마찬가지로 `j` 에 대한 나열이기 때문이다.

Repeat until convergence

{

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

}

구글에 검색하니까 1번으로 뜨는게 *vectorization(parallel computing)* 이더라. 병렬 연산에 많이 사용되나보다.

Refenrences

(1) StackExchange
(2) http://bt22dr.wordpress.com/
(3) http://spin.atomicobject.com

(4) http://www.holehouse.org/mlclass/

(5) http://www.longhaiqiang.com/